Final Project: Part 2
Integrating a Non-Blocking Cache Hierarchy with Out-of-Order Cores

1 Out-of-Order Cores

In the lectures and in the labs, you have only seen in-order processor pipelines. For this portion of the final project, you will be given an out-of-order (OoO) processor core, and you will add your cache hierarchy to a pair of these cores to create a multicore processor.

1.1 Register-Only Out-of-Order Core

The initial core provided for this lab is implemented with registers only, so there are no combinational paths between rules. Since there is a lot of interaction between many of the rules in the out-of-order processor, there is little to no concurrency between rules. The baseline assembly test runs in about 400 cycles, so that corresponds to a peak IPC of about 0.25. The TA is planning on releasing a concurrent version of the core before the end of the final project to reach IPC counts up to 1.0.

1.2 Out-of-Order Core Interface

The Out-of-Order core designed for non-blocking caches has an interface (`NBCore`) that is defined using the parametric interface `Core`. The `Core` interface has parametric types for instruction memory requests and responses and for data memory requests and responses. Although it is defined differently in the code, the `NBCore` interface is equivalent to the interface below:

```
interface NBCore;
    method ActionValue#(Tuple2#(CopRegIndex, Data)) cpuToHost;
    method Action hostToCpu(Addr startpc);
    interface ProcMemoryClient#(Addr, Data) iCacheClient;
    interface ProcMemoryClient#(NBCacheReq, NBCacheResp) dCacheClient;
    method Bool isRunning;
endinterface
```

The interface `ProcMemoryClient` is the complement of the parameterized memory interface `ProcMemory`. The definitions of each are shown below:

```
interface ProcMemory#( type req_type, type resp_type );
    method Action req( req_type req );
    method ActionValue#( resp_type ) resp;
endinterface

interface ProcMemoryClient#( type req_type, type resp_type );
    method ActionValue#(req_type) req;
    method Action resp( resp_type resp );
endinterface
```

As you can see, `ProcMemoryClient` has the same method names, but the methods are inverted; its `req` method outputs a request type and its `resp` method takes a response type as an input. The best way to think about `ProcMemoryClient#(a,b)` is that it is a user of `ProcMemory#(a,b)`.

In the example of the processor core, the core uses the non-blocking cache, so it has an interface that includes a `ProcMemoryClient` with types corresponding to the non-blocking cache. The interface for the non-blocking cache is equivalent to `ProcMemory` with request type `NBCacheReq` and response type `NBCacheResp`. To make connecting these interfaces together easy, there is an instance of the `Connectable` typeclass defined for them so the `mkConnection` module can be constructed to connect them together as follows:

```
ProcMemory#(a,b) mem;
ProcMemoryClient#(a,b) client;

// The below line constructs a module to connect mem to client
mkConnection(mem, client);
```

1.3 Target Memory Hierarchy

The initial code for the processor has a hierarchy of non-blocking instruction caches, but has no data cache hierarchy. You will have to implement your cache hierarchy in `Proc.bsv` and connect it to the cores and to the appropriate `WideMem` interface.

1.4 Using the Provided Code

The provided code is setup in a structure similar to labs 6 and 7, but there are a few different things to note:

- **incremental_build.sh is your friend!** Use `build -v` once to set up all the necessary files the first time, but after that, use the `incremental_build.sh` script in `src/scemi/` whenever you want to build the processor.
- Put the code for the four modules you constructed for Part 1 in `src/memory_hierarchy`.
- Some of the BSV files found in `src/includes/` have been changed from Part 1. They should still work the same way, but there has been some reorganization and some types have changed slightly.

2 Load-Linked and Store-Conditional

The load-linked (LL, also called load-reserve in lecture) and store-conditional (SC) instructions enable atomic read-modify-write (RMW) operations in the SMIPS ISA. These RMW operations that can be implemented with LL and SC include test-and-set, compare-and-swap, get-and-increment, and more.

Why do you need atomic RMW operations? Consider the case where you have two cores that are both trying to increment the same memory locations, such as a pointer to an array. If they use standard load and store instructions from the SMIPS ISA, the code would look like this for each core:

```
    # core 0                # core 1
    lw r1, 0(r2)            lw r1, 0(r2)
    addiu r1, r1, 1         addiu r1, r1, 1
    sw r1, 0(r2)            sw r1, 0(r2)
```

If the counter starts at 0 and the two cores perform the load before the other performs a store, then each core will write 1 as the incremented value.

How does LL and SC fix this problem and enable atomic RMW operations? By making the store conditional. The SC instruction only succeeds if the LL/SC pair and the code in between will appear to be atomic. If the SC instruction is successful, it will write 1 in the register that originally held the data to store. If the SC instruction fails, it will write 0 in that register. The following code will cause the shared counter to be incremented by each core once.

```
    # core 0                # core 1
st: ll r1, 0(r2)        st: ll r1, 0(r2)
    addiu r1, r1, 1         addiu r1, r1, 1
    sc r1, 0(r2)            sc r1, 0(r2)
    beq r1, r0, st          beq r1, r0, st
```

In the case that one of the cores fails, the process will restart.

Lets look into a specific timing for this code. The vertical position of each instruction corresponds to when the instruction was executed.

```
       # core 0                # core 1
1   st: ll r1, 0(r2)
2                           st: ll r1, 0(r2)
3       addiu r1, r1, 1
4                               addiu r1, r1, 1
5       sc r1, 0(r2)
6                               sc r1, 0(r2)
7       beq r1, r0, st
8                               beq r1, r0, st
```

After instruction 2 executes, both cores have the shared counter's cache line in state S in its cache. When instruction 5 executes, core 0 sees that the shared counter's cache line is still in the cache, so it hasn't been changed by any other cores, so it writes the incremented value to the counter's address and it writes 1 to r1 to denote success. Core 1 loses the cache line corresponding to the shared counter's cache when instruction 5 executes on core 0, so core 1 knows it can't perform an atomic operation with that address since someone modified it. When core 1 executes instruction 6, it doesn't perform the store and it writes 0 to r1 to denote failure. At this point, r1 is equal to r0, so instruction 8 will cause core 1 to try to increment again. Eventually core 1 will succeed in incrementing the shared counter.

These two examples are shown in the `incrementers` and `incrementers_atomic` benchmarks.

2.1 Implementation Details

We will be implementing LL and SC within the non-blocking cache. There are three things we need to add: a link address register for tracking cache lines corresponding to LL/SC instructions, logic for LL instructions, and logic for SC instructions.

Link Address Register: The link address register is a register in the non-blocking cache that tracks which cache line is being used in an active LL/SC pair. The type of this register should be `Maybe#(Addr)`. A valid value corresponds to an unbroken link that was started by a linked-load to that address (or an address in the same cache line). An invalid value corresponds to no link (no LL instructions have happened since the last SC instruction) or a broken link (a link for a cache line that has since been evicted from the cache).

When a cache line is evicted, its address is checked against the link address register. If there is a match, the link address register is set to invalid.

Logic for Load-Linked Instructions: When a load-linked instruction enters the cache (memory op is `Ll`), it is treated like a normal load instruction. The only difference is when the load-linked instruction returns a response, it writes the corresponding address to the link address register to mark an atomic operation in progress to that address.

Logic for Store-Conditional Instructions: When a store-conditional instruction enters the cache (memory op is `Sc`), it is treated like a normal store except it first checks the link address register. If the link address register does not match the incoming store-conditional instruction, it is dropped and a 0 is enqueued into the response queue to tell the processor the store failed. If it matches the link address register, then it is treated as a normal store, except if it hits and the store queue is empty, it also sends a 1 as a response.

When a store-conditional instruction makes it to the head of the store queue, it is checked against the link address register again to make sure the link is still valid before updating the memory or sending another upgrade request. Make sure you never update memory without checking the link address register, and make sure you never drop the instruction without sending a response, 0 for failure and 1 for success.

2.2 Testing Your LL and SC Implementation

There are a few provided programs that test your implementation of LL and SC. Unfortunately these programs use artificial memory accesses designed for an in-order core with a blocking cache. Because of this, **smipsv3_llsc.mc_asm.vmh will fail for your initial LL/SC implementation**. The test fails due to bypassing values from the store queue for a load-linked instruction effectively doing the LL instruction out of order with a store that will evict the corresponding cache line.

The best way to see your LL and SC implementations work are the two incrementers examples (part of the benchmark suite). One incrementer example is non atomic. For that one, the number of total increments

will not be the sum of each thread's number of increments. The other incrementer example *is* atomic. In that case it uses LL and SC instructions to make sure the total number of increments is the sum of each thread's number of increments.

3 Multicore Programs *(New Section – December 3rd)*

Multicore processors enable the use of multithreaded prorgams. Normally multithreaded programs manage thread creation using the OS. Since our processor has no OS, we write programs that can be run on multiple cores at the same time, and they perform different functions depending on the current core's core ID as read from the coprocessor.

3.1 Provided Multicore Programs

The provided code comes with four multicore assembly tests described below.

- `smipsv3_msi{1,2,3}.mc_asm.vmh` – Three multicore assembly tests for the MSI coherency protocol.
- `smipsv3_llsc.mc_asm.vmh` – A multicore assembly test for LL and SC designed for an in-order processor with non-blocking caches. This benchmark will fail. See the section *Testing Your LL and SC Implementation* for more information.

The provided code also comes with various multicore benchmark programs described below. One of the most interesting programs is `mc_multiply2.mc_bench.vmh` since it splits work across cores dynamically.

- `hello.mc_bench.vmh` – Multicore hello world example. Core 0 writes "Hello world!" and a description of the program into a software FIFO, and core 1 reads and prints from the software FIFO.
- `incrementers.mc_bench.vmh` – Both cores increment a shared counter 1000 times without using LL or SC. Since the memory operations are not atomic, the shared counter will not be 2000 when the program finishes.
- `incrementers_atomic.mc_bench.vmh` – Both cores increment a shared counter 1000 times atomically using LL and SC. Since the increment operations are atomic, the shared counter will be 2000 when the program finishes.
- `mc_median.mc_bench.vmh` – Multicore version of the median benchmark. Work is split between cores statically.
- `mc_multiply.mc_bench.vmh` – Multicore version of the multiply benchmark. Work is split between cores statically.
- `mc_multiply2.mc_bench.vmh` – Alternate multicore version of the multiply benchmark. This version splits the work to do dynamically by each core grabbing a chunk of operands to multiply whenever it is out of work to do.
- `mc_print.mc_bench.vmh` – A simple printing example. Core 0 prints that it is core 0, and all other cores print that they are not core 0.
- `mc_vvadd.mc_bench.vmh` – Multicore version of the vector addition benchmark. Work is split between cores statically.

3.2 Running Multicore Programs

The multicore programs are run as part of `run_assembly` and `run_benchmarks`. If you want to run a multicore program by itself you have to tell the `tb` program how many cores to use. The syntax for that is shown below:

```
./tb [NUMCORES] PROGRAMS... [NUMCORES PROGRAMS...]

# This example runs the single core multiply benchmark on one core and then
# runs the two multicore multiply benchmark on two cores
./tb 1 multiply.bench.vmh 2 mc_multiply.mc_bench.vmh mc_multiply2.mc_bench.vmh
```

If the number of cores is unspecified, `tb` assumes one core.

3.3 Custom Multicore Programs

To create your own multicore program, use one of the existing programs as a template. Assembly programs have to check the core ID on their own, but C benchmarks have the core ID passed in as an input to main. A typical main function for a multicore C programs looks like this:

```
int main( int coreid ) {
    if( coreid == 0 ) {
        return core0_program();
    } else if( coreid == 1 ) {
        return core1_program();
    }
    return 0;
}
```

If a variable is going to be shared between the two cores, it has to be defined as `volatile` so the compiler knows to generate code to read the variable from main memory whenever the processor wants to use the variable.

To compile your custom program, modify the makefile to include your program. If you don't know how to write makefiles, you should be able to follow the pattern for other programs.

4 Checkpoints

4.1 Checkpoint 1: Single-Thread Programs without LL/SC

To get single-threaded programs working, you have to instantiate your memory hierarchy in `Proc.bsv` and connect your non-blocking caches to the cache client subinterface in the processor core. Without LL and SC support, your processor will stall on all test programs that use LL and SC.

4.2 Checkpoint 2: Single- and Multi-Threaded Programs with LL/SC

Once you implement LL and SC correctly in your non-blocking cache, you will be able to use your processor on a suite of multi-threaded benchmarks and assembly programs. Read the subsection titled *Testing Your LL and SC Implementations* to see why you may be failing one of these tests.

4.3 Checkpoint 3: Add something *(More examples added – December 3rd)*

By this point you have a working multicore processor with non-blocking caches, but there is so much more you can do. This checkpoint challenges you to take your processor further in some direction. You could:

- Add a 4-way associative L2 cache to the parent protocol processor.
- Add an intermediate layer of caches to the cache hierarchy.
- Create a non-blocking parent protocol processor.
- Reduce the number clock cycles required for processing upgrade responses in the non-blocking cache and upgrade requests in the parent protocol processor.

- Create new benchmark programs that stress the non-blocking cache significantly.
- Suggest a solution to `smipsv3_llsc.mc_asm.vmh` failing and try it out.

4.4 Checkpoint 4: Make a Presentation *(Modified – December 3rd)*

On December 10th from 3 PM to 6 PM, we will have final presentations for this project and some pizza at the end. We would like you to prepare a 10 to 12 minute presentation about your final project. You should talk about what you did differently to your processor or memory hierarchy (checkpoint 3). If you got it working, you should explain what you did to implement it. If you did not get it working, don't worry, your presentation should present the progress you made, the difficulties you encountered, and what is left to do.